CSC 311 – Winter 2022-2023
Design and Analysis of Algorithms

3. Analysis of time efficiency of non-recursive algorithms

Prof. Mohamed Menai
Department of Computer Science
King Saud University

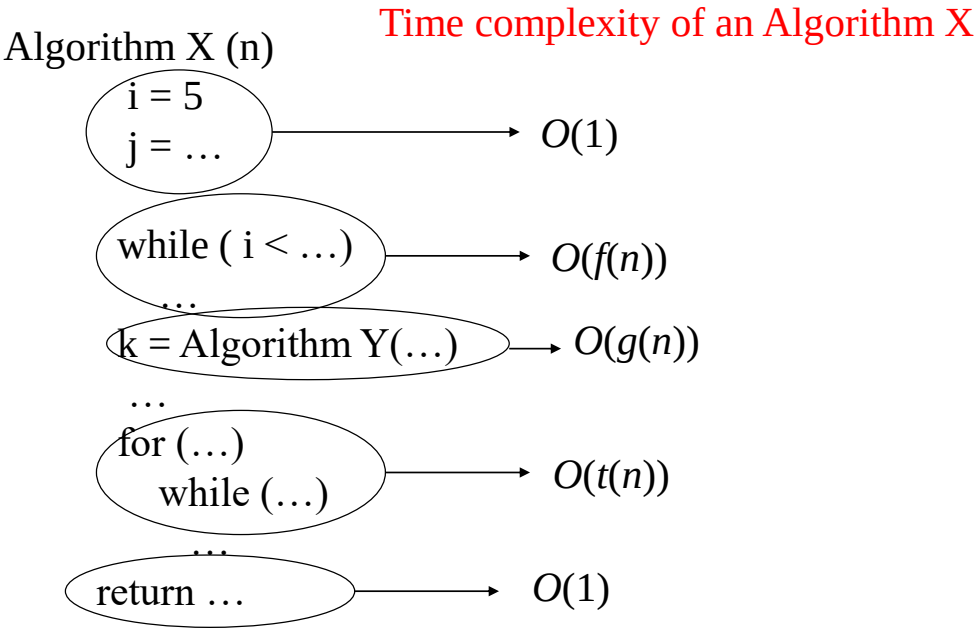
Outline

- Time efficiency of non-recursive algorithms
- Linear loops
- Logarithmic loops
- Nested loops
- Examples

Time efficiency of non-recursive algorithms

- Decide on parameter *n* indicating input size.
- Identify algorithm's basic operation.
- Determine worst, average, and best cases for input of size *n*.
- Sum the number of basic operations executed.
- Simplify the sum using standard formulas and rules.

Time efficiency of non-recursive algorithms



Input size and basic operation examples

Problem	**Input size measure**	**Basic operation**
Searching for key in a list of *n* items	Number of list's items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer *n*	*n*'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

5

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $T_{worst}(n)$ – maximum over inputs of size n
- Best case: $T_{best}(n)$ – minimum over inputs of size n
- Average case: $T_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs. So, avg = expected under uniform distribution.

6

Some standard formulas

$$\sum_{i=l}^{u} 1 = 1+1+\ldots+1 = (u-l)+1$$

$$\sum_{i=1}^{n} i = 1+2+\ldots+n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = 1^2+2^2+\ldots+n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} a^i = 1+a+a^2\ldots+a^n = \frac{a^{n+1}-1}{a-1}, \forall a \neq 1$$

7

Linear loops

Example 1: Assume that i is an integer.

$T(n) = O(n)$

```
i=1
Loop (i <=n)
        Application code
        i=i+1
```

Example 2:

```
i=1
Loop (i <=n)
        Application code
        i=i+2
```

Logarithmic loops

Multiply Loop

```
i=1
Loop (i < n)
        Application code
        i=i*2
```

Divide Loop

```
i=n
Loop (i > 0)
        Application code
        i=i / 2
```

$T(n) = O(\log n)$

Multiply → $2^{\text{iterations}} < n$ **Divide →** $n / 2^{\text{iterations}} >= 1$

Multiply (n=1000)		Divide (n=1000)	
Iteration	**Value of I**	**Iteration**	**Value of I**
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

9

Nested Loops

```
i=1
Loop (i < = n)
      j=1
      Loop (j<=n)
            Application code
            j=j*2
      i=i+1
```

$T(n) = O(n \log n)$

The number of iterations in the inner loop is logn. In the above program code, the inner loop is controlled by an outer loop. The above formula must be multiplied by the number of times the outer loop executes, which is n.

10

Nested Loops: Quadratic Time

```
i=1
   Loop (i < = n)
        j=1
        Loop (j<=n)
                Application code
                j=j+1
        i=i+1
```

$T(n) = O(n^2)$

The outer loop executed n times. For each of its iterations, the inner loop is also executed n times.

11

Nested Loops: Dependent Quadratic Time

```
i=1
Loop (i < = n)
        j=1
        Loop (j<=i)
                Application code
                j=j+1
        i=i+1
```

$$T(n) = n(n+1)/2 = O(n^2)$$

Here, the outer loop is same as the previous loop. However, the inner loop is dependent on the outer loop for one of its factors. It is executed only once the first iteration, twice the second iteration, three times the third iteration and so on. The number of iterations in the body of the inner loop is 1+2+3+4+……+8+9+10+…+n.

Example 1: Maximum element

```
MaxElement(A[0..n-1])
  maxval = A[0]
  for i = 1 to n-1 do
      if A[i] > maxval
          maxval = A[i]
  return maxval
```

$$T(n) = \sum_{1 \le i \le n-1} 1 = n - 1 = \Theta(n) \text{ comparisons}$$

13

Example 2: Element uniqueness problem

```
//Determines whether all
//elements in a given array
//are distinct

UniqueElement(A[0..n-1])
 for i = 0 to n-2 do
       for j = i+1 to n-1 do{
       if A[i] ==  A[j]
             return false}
  return true
```

$$
\begin{aligned}
T(n) &= \sum_{0 \le i \le n-2} \left(\sum_{i+1 \le j \le n-1} 1 \right) \\
&= \sum_{0 \le i \le n-2} (n-i-1) \\
&= (n-1) \sum_{0 \le i \le n-2} 1 - \sum_{0 \le i \le n-2} i \\
&= (n-1)^2 - (n-2)(n-1)/2 \\
&= n(n-1)/2 \\
&= O(n^2) \text{ comparisons}
\end{aligned}
$$

14

Example 3: Matrix multiplication

```
MatrixMult(A[0..n-1,0..n-1], B[0..n-1,0..n-1])
 for i = 0 to n-1 do
       for j = 0 to n-1 do
           C[i,j] = 0
            for k = 0 to n-1 do
                C[i,j] = C[i,j] + A[i,k] *B[k,j]
 return C
```

$$T(n) = \sum_{0 \le i \le n-1} \left(\sum_{0 \le j \le n-1} n \right)$$
$$= \sum_{0 \le i \le n-1} \Theta(n^2)$$
$$= \Theta(n^3) \text{ multiplications}$$

15

Example 4: Counting binary digits

Binary(n)
//Input: A positive decimal integer
//Output: The number of binary digits in n's binary representation
count = 1
while n > 1 **do**
    count = count + 1
    n = $\lfloor$n/2$\rfloor$
return count

Find integer i such that $n / 2^i \leq 1$.

Answer : $i \geq \log n$

So, $T(n) = \Theta(\log n)$ divisions.

16

Reading

Chapter 2 (Section 2.3)

Anany Levitin, Introduction to the design and analysis of algorithms, 3rd Edition, Pearson, 2011.